Web Performance Optimisation

Celine Florian, Stephan Robinig, Piotr Siewiera, and Nina Tschikof

706.041 Information Architecture and Web Usability WS 2025
Graz University of Technology

15 Dec 2025

Abstract

Web performance plays a critical role in user satisfaction, accessibility, and overall system efficiency. Even small delays in page load time can lead to increased bounce rates, reduced engagement, and negative business impact. Performance optimisations, such as asset minification, caching strategies, image compression, and efficient rendering techniques, help ensure fast, responsive user experiences across devices and network conditions. This survey describes numerous techniques to improve the performance of web sites and applications. The techniques are grouped into three categories: download performance, perceived performance, and runtime performance.

To illustrate these effects, a demo application compares two versions of the same web site: one operating with performance optimisations enabled, the other running without them. The demo highlights measurable differences in load time, resource consumption, and perceived responsiveness. By contrasting both implementations, the application demonstrates how targeted optimisation techniques can significantly improve real-world web performance.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Web performance improvements focus on making web sites faster, smoother and more responsive for users. A key area is download performance, which centres on reducing the amount of data transferred and speeding up the delivery of resources. Techniques such as compression, caching and reducing unnecessary network requests help ensure that pages begin to load as quickly as possible.

Another important area is perceived performance. This involves designing the loading experience, so that users see meaningful content early, even if the page is not fully ready. Strategies like prioritising above-the-fold content, using placeholders and loading non-critical elements later can greatly improve how fast a site feels.

Runtime performance deals with how efficiently a web site executes code once it is loaded. Reducing heavy JavaScript tasks, avoiding layout thrashing, and using modern APIs can help maintain smooth interactions and stable rendering.

All of these improvements align closely with Google's Web Vitals [CfD 2025c], a set of user centered metrics that measure loading speed, visual stability, and responsiveness. Focusing on these metrics helps ensure that performance work yields real benefits for the overall user experience.

Chapter 2

Web Vitals

Web Vitals provide standardised, user-centric metrics that quantify how quickly a site loads, how responsive it feels, and how stable its visual layout is. By focusing on measurable aspects of real user experience, they help developers identify performance issues that directly affect engagement, satisfaction, and retention. Since major search engines like Google also consider Web Vitals in their ranking algorithms, optimising these metrics not only improves usability, but can also enhance a site's visibility and overall success.

2.1 Core Web Vitals

Core Web Vitals are a set of three primary metrics standardised by Google to quantify real-world user experience on the web [CfD 2025c]:

- *Largest Contentful Paint (LCP)*: LCP measures loading performance by capturing the render time of the largest visible content element. A good user experience requires LCP to be less than or equal to 2.5 seconds. According to recent Chrome UX Report data, approximately 67.7% of pages meet this threshold [CfD 2025a].

- *Interaction to Next Paint (INP)*: INP measures interactivity by tracking responsiveness across user inputs. A good INP score is less than or equal to 200 ms. Roughly 85.9% of pages achieve this benchmark [CfD 2025a].

- *Cumulative Layout Shift (CLS)*: CLS quantifies visual stability by measuring unexpected layout shifts. A CLS score of less than or equal to 0.1 is considered good. Around 80.3% of pages reach this goal [CfD 2025a].

Across all Core Web Vitals, 54.4% of pages satisfy all three metrics simultaneously [CfD 2025a].

2.2 Further Web Vitals

Although the Core Web Vitals focus on key aspects of user experience, further metrics like Speed Index (SI) and Total Blocking Time (TBT) exist alongside them[CfD 2025b], to provide additional insight into loading behaviour and responsiveness:

- *Speed Index (SI)*: SI measures how quickly the contents of a page are visually populated. It is typically calculated through the "visual incompleteness" over time.

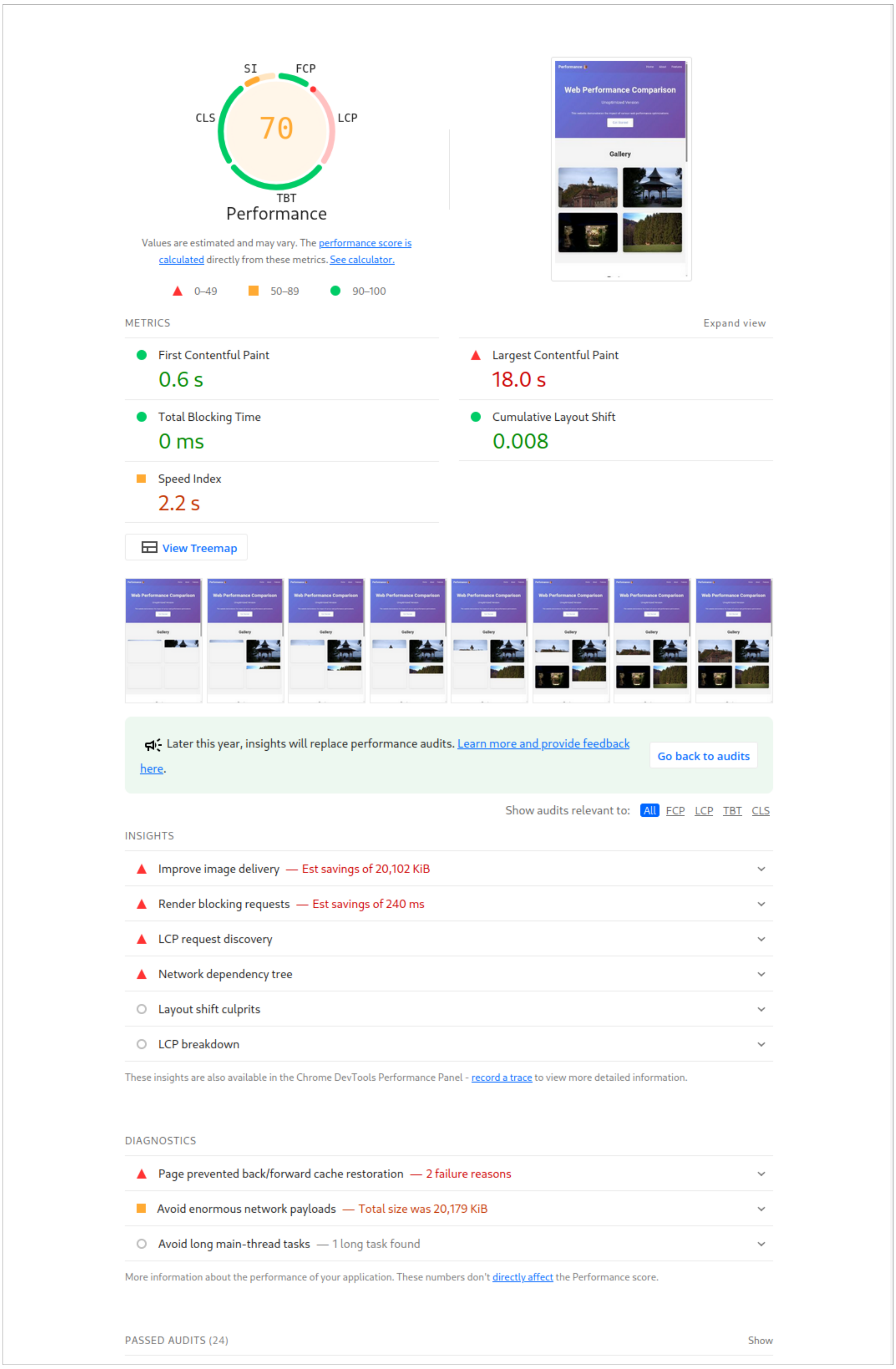
- *Total Blocking Time (TBT)*: TBT measures the amount of time during which the main thread is blocked and unable to respond to user input. It is computed as the sum of all long tasks whose duration exceeds 50 ms.

While not part of the official Core Web Vitals set, SI and TBT help highlight visual rendering speed and main-thread blocking issues, giving developers a more complete understanding of overall performance.

2.3 Auditing with Lighthouse

Lighthouse is an automated auditing tool designed to evaluate the quality and performance of web pages[CfD 2025b]. It provides comprehensive performance reporting that includes diagnostics and improvement suggestions across multiple categories. Importantly, Lighthouse also measures key user experience indicators such as the Core Web Vitals, offering insights into loading performance, interactivity, and visual stability. Integrated directly into the Chrome browser, Lighthouse allows developers to run audits easily and receive actionable recommendations to improve their sites.

A Lighthouse report provides an automated evaluation of a web page's quality across multiple categories, including performance, accessibility, best practices, SEO, and progressive web app capabilities. Since its measurements can be influenced by external factors such as network conditions, device performance, and background system load, it is recommended to run multiple tests to obtain reliable and stable results. The report includes a wide range of audit test cases and offers detailed, actionable feedback on how to address detected issues, enabling developers to systematically improve their site's performance and overall user experience. Figure 2.1 shows a Lighthouse performance report for the unoptimised web site created for this survey.



Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS

Expand view

● First Contentful Paint
0.6 s

▲ Largest Contentful Paint
18.0 s

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0.008

■ Speed Index
2.2 s

View Treemap

Later this year, insights will replace performance audits. Learn more and provide feedback here.

Go back to audits

Show audits relevant to: All FCP LCP TBT CLS

INSIGHTS

▲ Improve image delivery — Est savings of 20,102 KiB

▲ Render blocking requests — Est savings of 240 ms

▲ LCP request discovery

▲ Network dependency tree

○ Layout shift culprits

○ LCP breakdown

These insights are also available in the Chrome DevTools Performance Panel - record a trace to view more detailed information.

DIAGNOSTICS

▲ Page prevented back/forward cache restoration — 2 failure reasons

■ Avoid enormous network payloads — Total size was 20,179 KiB

○ Avoid long main-thread tasks — 1 long task found

More information about the performance of your application. These numbers don't directly affect the Performance score.

PASSED AUDITS (24)

Show

Figure 2.1: Lighthouse performance report for `https://stofflr.github.io/WebPerformanceComparison/output/unoptimized/index.html`.

Chapter 3

Improving Download Performance

Download performance focuses on reducing the amount of data transferred from web server to web client. Methods such as caching, minification, and file compression directly influence overall page load time and performance metrics such as Largest Contentful Paint (LCP).

3.1 Caching

Caching is a technique that improves web site performance by temporarily storing certain data on the client side. When a user revisits a page or navigates between pages, the browser can load previously cached resources instead of downloading them again from the server. As Alderson [2025] explains, this is an important technique since caching directly influences four fundamental aspects of how a site performs and scales:

- *Speed*: By providing files from the cache, the browser avoids additional requests over the network. Retrieving data already stored in memory happens almost instantly, whereas a new network request typically takes hundreds of milliseconds for the data to be returned. When many page elements can be loaded in this way, the entire web site feels faster, user interactions become smoother, and key performance metrics, such as Core Web Vitals, improve significantly.

- *Resilience*: Effective caching multiplies your system's capacity during high traffic periods. When a CDN or browser cache handles 80% of the load, the origin servers only have to handle the remaining 20%. This reduction in load helps prevent outages and ensures web site stability even during heavy traffic spikes, such as seasonal sales peaks or viral traffic.

- *Cost*: Serving resources from cache reduces the number of hits your origin has to process. CDN traffic is inexpensive, while uncached requests consume compute time, database operations, and more expensive outbound bandwidth. Even a slight increase in your cache hit rate can result in significant savings, especially when resources are cached directly in users' browsers and never reach the CDN at all.

- *SEO*: Search engines also benefit from caching. Effective caching reduces redundant crawling, allowing bots to focus on newer or deeper content. In addition, faster load times have a positive effect on search performance signals, which can improve overall visibility in search results.

Every browser maintains two types of cache: a memory cache and a disk cache. The memory cache lives only for the duration of an open tab and is optimised for speed, assets stored there can be reused almost instantly during the same page visit. Since this cache operates at an internal browser level, it can reuse files even when HTTP headers such as `no-store` would normally prohibit long-term caching [Alderson 2025].

The disk cache (HTTP cache) works very differently. It survives across multiple tabs, browser restarts, and future sessions, allowing it to retain far larger resources over time. This longer-lived cache follows the

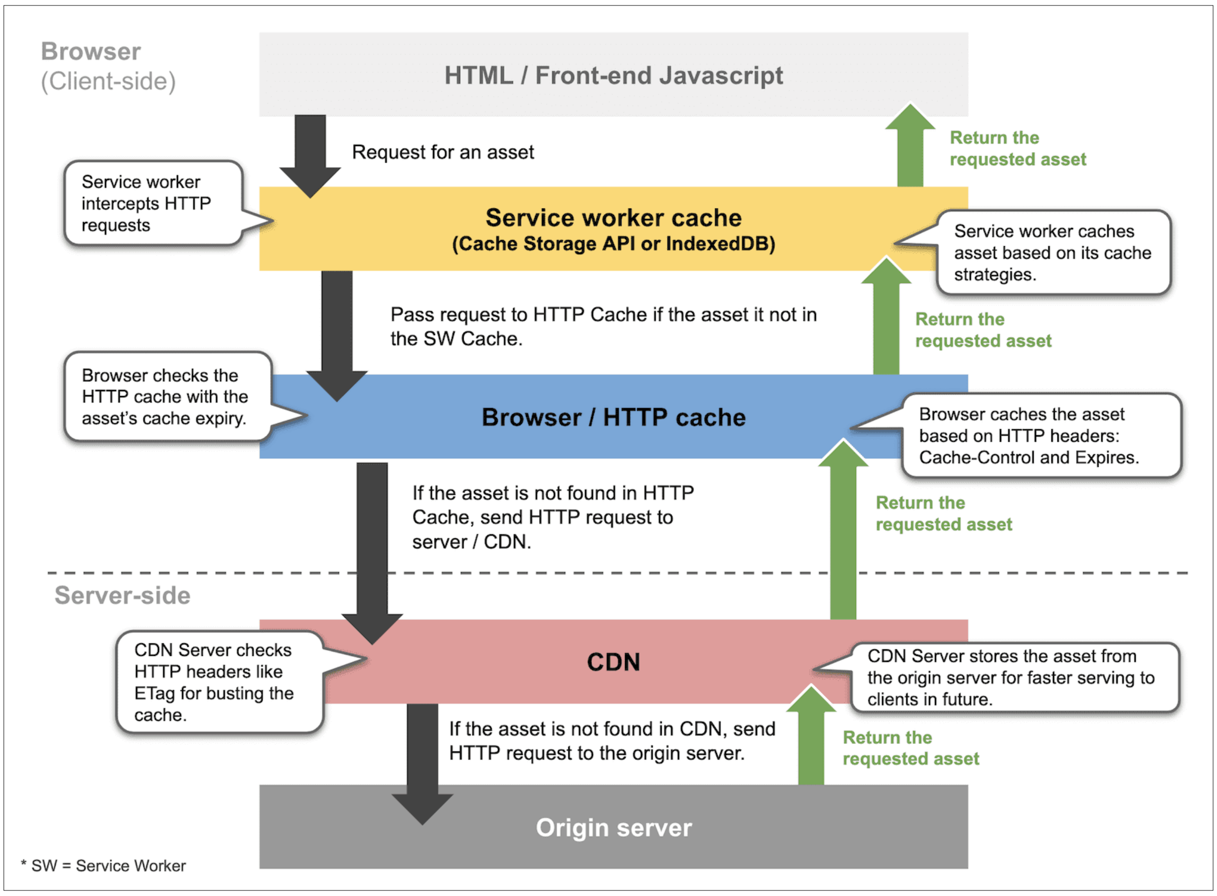

Browser
(Client-side)
HTML / Front-end Javascript
Request for an asset
Return the requested asset
Service worker intercepts HTTP requests
Service worker cache
(Cache Storage API or IndexedDB)
Service worker caches asset based on its cache strategies.
Pass request to HTTP Cache if the asset it not in the SW Cache.
Return the requested asset
Browser checks the HTTP cache with the asset's cache expiry.
Browser / HTTP cache
Browser caches the asset based on HTTP headers: Cache-Control and Expires.
If the asset is not found in HTTP Cache, send HTTP request to server / CDN.
Return the requested asset
Server-side
CDN Server checks HTTP headers like ETag for busting the cache.
CDN
CDN Server stores the asset from the origin server for faster serving to clients in future.
If the asset is not found in CDN, send HTTP request to the origin server.
Return the requested asset
Origin server
* SW = Service Worker

Figure 3.1: The flow through various types of caching between web server and web browser. [Image extracted from Chen [2025]. Used under the terms of a CC BY 4.0 licence.]

browser's interpretation of HTTP caching rules, using headers like `Cache-Control` and `Expires` when available, though the browser may still fall back on its own logic if those hints are missing or unclear [Alderson 2025]. A service worker cache can sit in front of the HTTP cache to implement custom caching logic. On the server side, a CDN can sit in front of the origin server to distribute assets geographically. This is shown in Figure 3.1.

The purpose of an far-future `Expires` header, as described by Souders [2007], is to tell the browser whether it should request a file from the server or simply load it from the cache. By giving the browser an explicit future date, the server signals that a resource should be considered fresh until that timestamp is reached. Files that rarely change, such as logos or versioned static assets, can be given long or "far-future" expiry dates while files that change frequently should be given shorter lifetimes. This reduces unnecessary HTTP requests and avoids re-downloading files that have not changed, which in turn improves load times and lowers server strain. Users benefit from faster repeat visits, and servers benefit from reduced load.

Historically, as stated by [GTmetrix 2025], `Expires` headers were the primary means of controlling freshness, but modern best practices rely on the more flexible `Cache-Control` header, which supports directives such as max-age, immutable, and must-revalidate. Nevertheless, the `Expires` header will continue to be widely supported for the foreseeable future and used alongside `Cache-Control` for maximum compatibility, especially for long-lived, versioned static assets.

The `Cache-Control` header functions as the central control panel for caching behaviour. The header, as described by [Alderson 2025], supports a rich set of directives that influence how both browsers and intermediary caches store and reuse resources. Its freshness directives include settings like `max-age` (how long a response stays fresh), `s-maxage` (a version of max-age that applies only to shared caches such as CDNs), `immutable` (indicating that the file will never change), `stale-while-revalidate` (permitting the use of outdated content while a new version is fetched), and `stale-if-error` (allowing stale content to be served when the origin fails). It also provides usage and storage directives, including `public` (cacheable anywhere), `private` (browser-only caching), `no-cache` (store but revalidate before use), no-store (do not cache at all), `must-revalidate` (stale responses must be validated with the origin), and `proxy-revalidate` (a similar rule applied specifically to shared caches).

Browsers can also send their own `Cache-Control` request headers, which guide the behaviour of caches along the network path. These directives don't override server rules, but they express what the client prefers. Examples include `no-cache` (request fresh validation before reuse), `no-store` (avoid caching entirely), and `only-if-cached` (return a cached response if one exists, otherwise fail, useful for offline operation). Additional directives like `max-age`, `min-fresh`, and `max-stale` let clients specify how strict they want to be about freshness or staleness.

Together with timestamps from the `Date` header, these directives allow caches, both browser and CDN, to calculate exactly when a resource becomes stale, when revalidation is required, and how aggressively a file can be reused. This level of precision makes `Cache-Control` the cornerstone of modern, reliable, and high-performance caching strategies.

While HTTP caching headers provide strong control over how resources are stored and reused, there are situations where even more precision or custom behaviour is needed. In these cases, developers can turn to Service Workers, which offer a level of control far beyond what traditional caching mechanisms can provide.

A Service Worker, as described by Lin [2024], is a background script that operates independently of the main browser thread. Once registered, it inserts itself between your web application and the network, acting as a programmable layer that can intercept requests, modify how responses are handled, and decide exactly when and how resources are cached. This architecture enables capabilities such as offline access, background synchronisation, and push notifications, features that are not possible with HTTP headers alone.

Service Workers follow a structured lifecycle consisting of several stages. During registration, the browser becomes aware of the Service Worker. In the installation phase, developers commonly pre-cache essential files so they are available even if the user goes offline. After installation comes activation, where outdated caches can be removed or migrated. Finally, during the fetch event, the Service Worker intercepts network requests, giving developers full control over whether to return a cached entry, retrieve a fresh version from the network, or combine both strategies. A Service Worker can even serve a fully cached version of a page when the network is unavailable.

This level of control enables sophisticated caching patterns, such as applying different strategies to different file types, using stale-while-revalidate behaviour even when the server does not support it, or implementing offline-first workflows for critical parts of an application. The result is improved performance, greater resilience, and enhanced reliability in poor or unstable network conditions.

However, Service Workers do introduce additional complexity. They require careful design to manage updates, avoid stale assets lingering in the cache, and work within browser-specific storage limits. When used thoughtfully, though, they provide a powerful extension to standard HTTP caching, making advanced performance optimisations and offline experiences possible.

3.2 Use HTTP/3

HTTP (Hypertext Transfer Protocol) is the foundation of communication on the web. It defines how browsers request resources and how servers deliver responses such as HTML files, images, scripts, or metadata. When someone visits a web site, their browser initiates an HTTP request, and the server responds with the appropriate files along with headers that describe capabilities, including which HTTP protocol versions are supported. The protocol version used for any given session depends on what both

HTTP	1.x	2.x	3.x
Usage	9.2%	60.4%	30.4%

Table 3.1: HTTP version usage. Data from 15 Nov 2025 [Monus 2025].

sides support. During the initial connection, the browser and server negotiate the highest available HTTP version they can both use. If a server supports HTTP/3, for example, a visitor may receive the web site over HTTP/3, HTTP/2, or even HTTP/1.1, depending on their browser's capabilities.

Supporting modern HTTP versions is important for performance and SEO. Newer protocols reduce connection latency, improve reliability, and handle multiple resource transfers more efficiently. HTTP/2 offers multiplexing, allowing the browser to request several files in parallel without the bottlenecks of HTTP/1.1. HTTP/3, built on the QUIC transport layer, establishes connections faster and is more resilient to packet loss, resulting in quicker resource downloads and improved metrics such as Largest Contentful Paint (LCP) [Monus 2025]. These improvements contribute to smoother page loads, better Core Web Vitals, and ultimately stronger search visibility.

Although HTTP/3 adoption is growing rapidly, it is far from universal, and HTTP/2 remains the dominant protocol. Additionally, some widely used browsers, such as Samsung Internet, Opera Mini, QQ Browser, and KaiOS Browser, still do not support HTTP/3 [Caniuse 2025]. Without fallback, these users would experience degraded performance or fail to connect at all. For this reason, the recommended approach is to enable HTTP/3 while providing a reliable fallback to HTTP/2. This ensures that users with modern browsers benefit from the improved speed and resilience of HTTP/3, while all other visitors continue to receive a stable and fully functional experience through the more widely supported HTTP/2 protocol.

3.3 Ship Fewer Bytes

The strategy to "Ship Fewer Bytes" directly targets improving download performance by reducing the total amount of data that must be transferred over the network from the server to the client. This is achieved through optimising code, minimising assets, and compressing files.

A primary and powerful technique for reducing the JavaScript payload size is *tree shaking*. Tree shaking is a specialised process for eliminating unused or dead code from JavaScript bundles during the application build process [DEV 2025]. The name itself is a metaphor: the application's code is viewed as a tree, and by "shaking" it, the unnecessary branches (the unused code) fall off and are removed from the final bundle. The process is heavily reliant on the static analysis of module dependencies, which is only enabled by modern ES6 (import and export) module syntax. Since the connections between modules are fixed and known at build time, bundlers like Webpack or Rollup can reliably trace the execution path and safely eliminate code that is imported but never actually executed. This ensures that only the necessary "live" code is included in the production bundle, leading to significantly smaller file sizes and faster application load times.

Once the code base is optimised to contain only live code, the remaining text assets must be made as small as possible before transmission. This involves minification, which should be applied to all text assets, including HTML, SVG, CSS, and JS. Finally, all text assets must be served using modern compression techniques.

Data 1GB	Size Reduction	Compression Speed	Decompression Speed
Brotli	32%	0.7 Kb/s	380 Mb/s
Gzip	23%	27 Mb/s	270 Mb/s

Table 3.2: Results of a compression experiment.

3.4 File Compression

A critical method to speed up the download time of the resources is enabling file compression on the server. By compressing text assets, HTML, CSS and JS files before transmission, data size is reduced, leading to faster retrieval on the client side. As of 2025, two major algorithms used for file compression are Brotli and Gzip, with Brotli being the more effective [Evans 2022]. It is recommended to enable Brotli (br) compression and configure a fallback to Gzip for maximum compatibility. This compression step further reduces the file size just prior to network transmission, making the data transfer phase as quick as possible.

An experiment was conducted compressing a textual dataset from Mahoney [2011]. The dataset was compressed with both Brotli and Gzip. The results are shown in Table 3.2 and confirm that Brotli achieves a 32% size reduction, outperforming Gzip's 23%. While Gzip compresses approximately 4000% faster (27 Mb/s vs. 0.7 Kb/s), Brotli decompresses 37% faster (380 Mb/s vs. 270 Mb/s). Given that compression occurs once on the server, but decompression impacts every user visit, Brotli is recommended as the primary compression method, with Gzip as a fallback.

3.5 Minification

Minification, as described by Anderson [2025], is a technique performing a variety of code optimisations to scripts or CSS, such as removing unnecessary white space, comments, or shortening variable names to reduce the size of the files. While this technique helps with file size and thus download speed, it has its drawbacks, as the minified file has reduced readability and debug capability.

The technique results is presented in the attached code listings. Listing 3.1 presents original, unminified code, while Listing 3.2 displays the minified version of the same code. As can be seen, the variable names are reduced to minimum and comments and unnecessary white space were removed.

3.6 Other Techniques

Additional strategies further enhance download performance include:

- Using modern image formats, such as WebP and AVIF, as described by Salvado [2025]. They provide superior compression efficiency over traditional JPEG and PNG. WebP files are typically 25-34% smaller than JPEGs at comparable quality, while AVIF can be even 20% smaller than WebP. Both formats are widely supported and can be delivered via the `<picture>` element.

- Using `srcset` and `sizes` ensures appropriate image resolution are served, i.e. for mobile users, reducing unnecessary data transfer, while setting explicit width and height attributes prevents layout shifts during image loading.

- Bundling, as described by Anderson [2025], combines multiple JavaScript or CSS files into single files, thus reducing the number of HTTP requests required to retrieve the web site's data, potentially improving load times. However, this approach introduces a cache invalidation trade-off. When any single resource within the bundle changes, the entire bundle must be re-downloaded.

```
function addNumbers(a, b) {
  const result = a + b;
  return result;
}

function multiplyNumbers(a, b) {
  // Handle edge cases
  if (b === 0 || a === 0) {
    return 0;
  }

  // Handle negative numbers
  const isNegative = (a < 0 && b > 0) || (a > 0 && b < 0);
  const absA = Math.abs(a);
  const absB = Math.abs(b);

  let result = 0;

  for (let i = 0; i < absB; i++) {
    result = addNumbers(result, absA);
  }

  // Apply sign if needed
  if (isNegative) {
    result = -result;
  }

  return result;
}
```

Listing 3.1: Unminified JavaScript code.

```
function addNumbers(e,r){return e+r}
function multiplyNumbers(e,r){if(0===r||0===e)return 0;
const n=e<0&&r>0||e>0&&r<0,t=Math.abs(e),o=Math.abs(r);let u=0;
for(let e=0;e<o;e++)u=addNumbers(u,t);return n&&(u=-u),u}
```

Listing 3.2: Minified JavaScript code.

- Content Delivery Networks (CDNs), as described by Susnjara and Smalley [2025], distribute assets across geographically dispersed servers. This leads to a faster delivery of data, as the web site looks for the geographically closest server to retrieve the data, thus reducing round trip time.

Chapter 4

Improving Perceived Performance

Perceived performance is the user's experience of speed. By strategically prioritising critical resources and deferring non-essential ones, developers can create the illusion of faster loading, even before all the assets have been retrieved.

4.1 Load What Matters First

This point focuses mostly on two suggestions: placing CSS in the `<head>` element and placing non-critical JavaScript at the bottom of the `<body>` element. The first suggestion downloads and loads CSS immediately when a new HTML file starts getting processed. This way, instead of blocking the parser to download stylesheet, the application already has the information and can continue rendering the website. The second method makes sure that the JavaScript does not block the parser if it is not immediately needed.

4.2 Manage Non-Critical Resources

Non-critical resources are resources that are not required for the initial rendering of the content. There are several ways to manage non-critical resources. Firstly, one can use `async` and `defer` attributes. The first ensures that the JavaScript resource is loaded asynchronously in the background, without blocking the rendering, while the second asks the browser to run the script after the rest of the document has been parsed.

Furthermore, it is important to understand the difference between the `preload` and `prefetch` attributes, as described in Osmani [2017], to know when to use them. The `preload` attribute is an early fetch instruction to the browser to request a resource needed for a page, whereas `prefetch` tells the browser to fetch the resources that will be required for the next page of the website, thus letting the critical resource requests be completed in parallel.

Lastly, it is important to lazy load images that are off-screen, as described by MDN [2025b], so as not to use browser's resources loading images that the user cannot see yet. Lazy loading is done via the attribute `loading="lazy"`.

Chapter 5

Improving Runtime Performance

Runtime performance focuses on execution speed, responsiveness to user input, and smooth visual updates while the application is running. Poor runtime performance can degrade user experience significantly. For this reason, reducing unnecessary workload and optimising JavaScript execution during runtime are important aspects of modern web performance optimisation.

5.1 Reduce Main-Thread Work

By default, the browser's main thread in the renderer process is responsible for handling most critical tasks. These tasks include parsing HTML to construct the Document Object Model (DOM), parsing and applying CSS styles, as well as parsing, evaluating and executing JavaScript code. In addition, the main thread processes user input events, such as clicks, scrolling, and keyboard interactions. Whenever the main thread is occupied with long-running, more complex tasks the browser may not be able to respond immediately to user interactions. This can result in interfaces that are frozen or temporary unresponsiveness, which has a negative impact on interactivity.

One possibility to avoid these problems is to use CSS instead of JavaScript, wherever equivalent functionality is available. This helps improve performance, since CSS is implemented natively in the browser and is highly optimised [Stack Overflow 2014]. JavaScript on the other hand is less error-tolerant and runs on the main thread, which can block rendering and other user interactions.

Another important technique for reducing main-thread work is the use of Web Workers. Web Workers allow JavaScript code to execute in background threads that are separated from the main thread. They are particularly suitable for performing complex tasks such as calculations or data processing [MDN 2025c]. By offloading these tasks to Web Workers, the main thread remains available for rendering and handling user interactions which results in a smoother and more responsive user interface.

5.2 Optimise JavaScript Runtime Performance

In addition to reducing the workload on the main thread, runtime performance can be improved by optimising JavaScript execution itself. Efficient JavaScript code reduces execution time, avoids unnecessary processing and improves responsiveness during user interaction. Writing efficient algorithms is a fundamental optimisation strategy. This includes avoiding unnecessary loops, minimising computational complexity, and preventing redundant calculations. Memory management plays a crucial role in long-running applications, especially single-page applications which remain active for extended periods. Improper handling of the memory can cause applications to gradually consume more and more resources and slow down over time. By carefully managing memory usage and releasing unused resources, applications can maintain stable and consistent performance throughout an entire user session.

Chapter 6

Demo

To illustrate the effect of various optimisation on web performance, two versions of the same web site were built, one unoptimised and one optimised. A Python script using argparse enables or disables various optimisation techniques and produces both variants for analysis. Both sites are self-hosted using separate `npm http-server` instances and a CloudFlare Reverse Proxy, allowing full control over HTTP protocol settings, caching behaviour, and other server-side parameters. Through the Python-based generation process, individual optimisation features can be selectively activated or deactivated, making it possible to analyse how each technique affects performance and resource usage. Hosting both variants on identical hardware ensures that differences in measurement are solely due to the applied optimisations rather than environmental factors.

6.1 Setup and Hardware

The web sites are both physically hosted on the same Raspberry Pi Zero 2W (64-bit), running Raspberry Pi OS Trixie[1] . Two independent `npm http-server` processes serve the optimised and unoptimised sites simultaneously from the same device. Performance data is collected using Cloudflare synthetic Real-time User Monitoring (RUM) [Cloudflare 2025b] and Lighthouse [CfD 2025b]. All related source code and configuration files are available in the project repository on GitHub [Robinig et al. 2025c].

6.2 Cloudflare Zero Trust Reverse Proxy

Cloudflare Zero Trust provides a secure reverse-proxy layer that enables the exposure of internal services to the public internet without requiring inbound port forwarding on the local network [Cloudflare 2025a]. By installing the Cloudflare Tunnel client (cloudflared) on the host system, an encrypted outbound connection to Cloudflare's network is established. This tunnel transports all external requests to the internal web server, meaning that no router configuration or open ports are necessary.

In practice, the setup involves authenticating the local machine with a Cloudflare account, defining one or more public hostnames within the Zero Trust dashboard, and binding these hostnames to the corresponding internal service endpoints. In this case `localhost:8080` and `localhost:8081` were used for the optimised and unoptimised versions of the web site, respectively. Once the tunnel is active, Cloudflare handles TLS termination, routing, access control and optional Zero Trust policies such as identity authentication or device posture checks. This approach provides a secure, maintenance-friendly method for exposing development or production services while maintaining a strictly closed local network.

[1]Build version 2025-10-02

6.3 Cloudflare RUM and Required CORS Configuration

Cloudflare's RUM provides passive performance insights by injecting a lightweight JavaScript snippet into responses served through Cloudflare [Cloudflare 2025b]. This script measures real user performance metrics (such as page load times, network latency, and interaction delays) and reports them back to Cloudflare's analytics platform.

To ensure that the injected RUM JavaScript can load and send data successfully, the origin server must allow cross-origin requests from Cloudflare's domains. This typically requires configuring the origin to send permissive CORS headers for the RUM endpoints. In many setups, Cloudflare automatically adds the necessary headers during script injection. However, if the origin applies strict CORS policies (for example, through framework or server middleware), Cloudflare-controlled origins must specifically be allowed, so the injected RUM script can execute and report metrics without being blocked by the browser. [MDN 2025a]

6.4 HTTP-Server

The `http-server` tool is a lightweight, zero-configuration command-line HTTP server available through `npm` [Node 2021]. It allows developers to quickly serve static files from any local directory, making it useful for testing, prototyping, and simple development workflows. A server can be started by running `npx http-server` in the target directory, providing immediate access to the files through a local web address.

6.5 Deployment Overview

This project generates two versions of a web site, an unoptimised and an optimised build, to demonstrate the impact of modern web performance techniques. The optimised variant includes:

- *Minification*: Removes whitespace and comments from HTML, CSS, JS.
- *Inline CSS / JS*: Eliminates render-blocking requests.
- *Deferred JS*: Adds the `defer` attribute.
- *Lazy Loading Images*: Uses `loading="lazy"`.
- *Fetch Priority*: Adds `fetchpriority="high"` to images.
- *Resource Hints*: Preconnect and DNS-prefetch for faster network setup.
- *Prefetch Hints*: Anticipatory loading for next page navigation.
- *Unused Code Removal*: Purges unused CSS/JS.
- *Image Optimisation*: Generates responsive sizes (`200w-1600w`) and compressed `.gz/.br` versions.

The settings are summarised in Table 6.1.

For consistent performance testing, both versions are served locally using npm's lightweight static server:

```
npm install -g http-server
```

The project provides a convenience script `serve.sh` that launches:

- *Optimised site*: at `http://localhost:8080`, configured to serve pre-compressed assets such as `.gz` (Gzip) and `.br` (Brotli), when available.
- *Unoptimised site*: at `http://localhost:8081` with caching disabled.

Setting	Unoptimised Site	Optimised Site
Minification (HTML/CSS/JS)	⁄	✓
Inline CSS / JS	⁄	✓
Deferred JS (`defer` attribute)	⁄	✓
Lazy Loading Images (`loading="lazy"`)	⁄	✓
Fetch Priority (`fetchpriority="high"`)	⁄	✓
Resource Hints (preconnect, dns-prefetch)	⁄	✓
Prefetch Hints (next page anticipation)	⁄	✓
Unused Code Removal (CSS/JS purge)	⁄	✓
Image Optimisation (responsive sizes, gz/br)	⁄	✓

Table 6.1: Unoptimised vs. optimised sites: Comparison of optimisation settings.

Although `http-server` does not compress files on the fly, it will automatically deliver pre-compressed assets generated by the script, for example `file.*.gz` or `file.*.br`. Cache behaviour may be adjusted explicitly:

```
http-server ./output/optimized --cache 3600
```

CORS can be enabled for cross-origin testing:

```
http-server ./output/optimized --cors
```

To generate both sites with full optimisations, the following command is used:

```
python generate_websites.py --all
```

To enable individual optimisations explicitly, command-line options can be provided:

```
python generate_websites.py --minify --inline-css --lazy-loading
```

This environment allows controlled measurement of how caching, compression, resource hints, and asset optimisations improve real-world performance. Note that both sites must use the same version of HTTP, by default HTTP/3 is used, with an option to change to HTTP/2. HTTP/1 cannot be used. Protocol setting is shared across both sites and controlled via Cloudflare.

6.6 Performance Comparison

The perfomance of the two sites can be assessed in two ways:

1. Browser DevTools (Network timing, payload size).

2. Lighthouse audits for rendering, interactivity, and LCP improvements.

For this comparison, Lighthouse reports were generated for two sites: unoptimised [Robinig et al. 2025b] and optimised [Robinig et al. 2025a]. Performance metrics may vary depending on network conditions, device performance, caching, and other local factors. For reliable results, multiple runs should be performed. The Lighthouse results are summarised in Table 6.2.

Metric	Unoptimised Site	Optimised Site
Largest Contentful Paint (LCP)	18.0 s	0.7 s
First Contentful Paint (FCP)	0.6 s	0.3 s
Cumulative Layout Shift (CLS)	0.008	0
Speed Index	2.2 s	0.3 s
Total Blocking Time (TBT)	0 ms	0 ms

Table 6.2: Unoptimised vs. optimised sites: Comparison of web vitals from Lighthouse audits.

Bibliography

Alderson, Jono [2025]. *A complete guide to HTTP caching*. 25 Aug 2025. `https://jonoalderson.com/performance/http-caching/` (cited on pages 7–8).

Anderson, Rick [2025]. *Bundling and Minification*. Microsoft, 06 Dec 2025. `https://learn.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification` (cited on page 11).

Caniuse [2025]. *HTTP/3 protocol*. 11 Oct 2025. `https://caniuse.com/?search=http+3` (cited on page 10).

CfD [2025a]. *Chrome UX Report Release Notes — October 2025*. Chrome for Developers, 11 Sep 2025. `https://developer.chrome.com/docs/crux/release-notes#202510` (cited on page 3).

CfD [2025b]. *Lighthouse Overview*. Chrome for Developers, 11 Sep 2025. `https://developer.chrome.com/docs/lighthouse/overview` (cited on pages 3–4, 17).

CfD [2025c]. *Understanding Core Web Vitals and Google Search Results*. Chrome for Developers, 12 Mar 2025. `https://developers.google.com/search/docs/appearance/core-web-vitals` (cited on pages 1, 3).

Chen, Jonathan [2025]. *Service worker caching and HTTP caching*. 07 Dec 2025. `https://web.dev/articles/service-worker-caching-and-http-caching` (cited on page 8).

Cloudflare [2025a]. *Cloudflare Zero Trust: Secure Tunneling and Reverse Proxy*. 2025. `https://cloudflare.com/zero-trust/products/access/` (cited on page 17).

Cloudflare [2025b]. *RUM Beacon for Web Analytics*. 2025. `https://developers.cloudflare.com/speed/observatory/rum-beacon/` (cited on pages 17–18).

DEV [2025]. *Understanding Tree Shaking in JavaScript: A Complete Guide*. 07 Dec 2025. `https://dev.to/softheartengineer/tree-shaking-in-js-51do` (cited on page 10).

Evans, Cal [2022]. *The Difference Between Brotli And Gzip Compression Algorithms To Speed Up Your Site*. SiteGround, 09 Feb 2022. `https://siteground.com/academy/brotli-vs-gzip-compression/` (cited on page 11).

GTmetrix [2025]. *YSlow: Add Expires headers*. 2025. `https://gtmetrix.com/add-expires-headers.html` (cited on page 8).

Lin, Dylan [2024]. *Optimizing Frontend Caching with Service Workers and Cache Strategies*. 18 Aug 2024. `https://medium.com/@ddylanlinn/optimizing-frontend-caching-with-service-worker-and-cache-strategy-4131ae1d9aa8` (cited on page 9).

Mahoney, Matt [2011]. *About the Test Data*. 01 Sep 2011. `https://mattmahoney.net/dc/textdata.html` (cited on page 11).

MDN [2025a]. *HTTP Access Control (CORS)*. Mozilla Developer Network, 30 Nov 2025. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS` (cited on page 18).

MDN [2025b]. *Lazy loading*. Mozilla Developer Network, 04 Nov 2025. `https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Lazy_loading` (cited on page 13).

MDN [2025c]. *Using Web Workers*. Mozilla Developer Network, 11 Sep 2025. `https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Using_web_workers` (cited on page 15).

Monus, Anna [2025]. *HTTP/3 vs HTTP/2 Performance: Is the Upgrade Worth It?* 07 Dec 2025. `https://debugbear.com/blog/http3-vs-http2-performance` (cited on page 10).

Node [2021]. *http-server*. 2021. `https://npmjs.com/package/http-server` (cited on page 18).

Osmani, Addy [2017]. *Preload, Prefetch And Priorities in Chrome*. 27 Mar 2017. `https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf` (cited on page 13).

Robinig, Stephan, Celine Florian, Piotr Siewiera and Nina Tschikof [2025a]. *Lighthouse Performance Report: Optimised Site*. 2025. `https://stofflr.github.io/WebPerformanceComparison/output/optimized/page2.html` (cited on page 19).

Robinig, Stephan, Celine Florian, Piotr Siewiera and Nina Tschikof [2025b]. *Lighthouse Performance Report: Unoptimised Site*. 2025. `https://stofflr.github.io/WebPerformanceComparison/output/unoptimized/page2.html` (cited on page 19).

Robinig, Stephan, Celine Florian, Piotr Siewiera and Nina Tschikof [2025c]. *Web Performance Comparison*. 2025. `https://github.com/StofflR/WebPerformanceComparison` (cited on page 17).

Salvado, Daniel [2025]. *AVIF vs. WebP: Speed, Quality, and Browser Support*. Crystallize, 03 Jun 2025. `https://crystallize.com/blog/avif-vs-webp` (cited on page 11).

Souders, Steve [2007]. *High Performance Web Sites*. O'Reilly, 01 Sep 2007. ISBN 0596529309 (cited on page 8).

Stack Overflow [2014]. *Is it always better to use CSS when possible instead of JS?* 03 Jun 2014. `https://stackoverflow.com/questions/24012569` (cited on page 15).

Susnjara, Stephanie and Ian Smalley [2025]. *What is a Content Delivery Network (CDN)?* IBM, 06 Dec 2025. `https://ibm.com/think/topics/content-delivery-networks` (cited on page 12).